**Assignment-1**

**1. Write a C program to implement the concept of Bubble sort on the above data set. Print the data set after every iteration.**

#include<stdio.h>

int main(){

int arr[]={27,15,39,21,28,70};

int i,j,swap;

printf("Original Dataset: ");

for(i=0;i<6;i++){

printf("%d ",arr[i]);

}

printf("\n");

for(i=0;i<6;i++){

for(j=i+1;j<6;j++){

if(arr[i]>arr[j]){

swap=arr[i];

arr[i]=arr[j];

arr[j]=swap;

}

}

printf("After %d iteration: ",i);

for(int k=0;k<6;k++){

printf("%d ",arr[k]);

}

printf("\n");

}

printf("After sorting: ");

for(i=0;i<6;i++){

printf("%d ",arr[i]);

}

return 0;

}

**2. Write a C program to implement the concept of Selection sort on the above data set. Print the data set after every iteration.**

#include<stdio.h>

int main() {

int arr[] = {27, 15, 39, 21, 28, 70};

int i, j, swap, min;

printf("Original Dataset: ");

for(i = 0; i < 6; i++) {

printf("%d ", arr[i]);

}

printf("\n");

for(i = 0; i < 6; i++) {

min = i;

for(j = i + 1; j < 6; j++) {

if(arr[j] < arr[min]) {

min = j;

}

}

if(min != i) {

swap = arr[i];

arr[i] = arr[min];

arr[min] = swap;

}

printf("After %d iteration: ", i);

for(int k = 0; k < 6; k++) {

printf("%d ", arr[k]);

}

printf("\n");

}

printf("After sorting: ");

for(i = 0; i < 6; i++) {

printf("%d ", arr[i]);

}

return 0;

}

**3. Write a C program to implement the concept of Insertion sort on the above data set. Print the data set after every iteration.**

#include <stdio.h>

void insertionSort(int arr[], int n) {

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

printf("After iteration %d: ", i);

for (int k = 0; k < n; k++) {

printf("%d ", arr[k]);

}

printf("\n");

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

insertionSort(arr, n);

printf("Sorted array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

**4. Write a C program to implement the concept of Quick sort on the above data set. Print the data set after every iteration**

#include <stdio.h>

void swap(int\* a, int\* b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

printf("After partitioning around %d: ", arr[pi]);

for (int k = low; k <= high; k++) {

printf("%d ", arr[k]);

}

printf("\n");

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

quickSort(arr, 0, n - 1);

printf("Sorted array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

**5. Write a C program to implement the concept of Merge sort on the above data set. Print the data set after every iteration.**

#include <stdio.h>

void merge(int arr[], int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

printf("After merging: ");

for (int x = l; x <= r; x++)

printf("%d ", arr[x]);

printf("\n");

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

mergeSort(arr, 0, n - 1);

printf("Sorted array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

**6. Write a C program to show that Quick sort is better than Bubble sort.**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void swap(int\* a, int\* b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void bubbleSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap(&arr[j], &arr[j + 1]);

}

}

}

}

int main() {

srand(time(NULL));

int n = 10000;

int arr1[n], arr2[n];

for (int i = 0; i < n; i++) {

arr1[i] = rand() % 10000;

arr2[i] = arr1[i];

}

clock\_t start, end;

start = clock();

quickSort(arr1, 0, n - 1);

end = clock();

printf("Time taken by Quick Sort: %lf seconds\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);

start = clock();

bubbleSort(arr2, n);

end = clock();

printf("Time taken by Bubble Sort: %lf seconds\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);

return 0;

}

**7. Write a C program to show that merge sort is more effective than quick sort**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void merge(int arr[], int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

srand(time(NULL));

int n = 10000;

int arr1[n], arr2[n];

for (int i = 0; i < n; i++) {

arr1[i] = rand() % 10000;

arr2[i] = arr1[i];

}

clock\_t start, end;

start = clock();

mergeSort(arr1, 0, n - 1);

end = clock();

printf("Time taken by Merge Sort: %lf seconds\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);

start = clock();

quickSort(arr2, 0, n - 1);

end = clock();

printf("Time taken by Quick Sort: %lf seconds\n", ((double)(end - start)) / CLOCKS\_PER\_SEC);

return 0;

}

**Assignment-2**

**1. Write a C program to create a binary tree using recursive function and display that level wise.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (root->left == NULL) {

root->left = insertNode(root->left, data);

} else if (root->right == NULL) {

root->right = insertNode(root->right, data);

} else {

root->left = insertNode(root->left, data);

}

return root;

}

void printGivenLevel(struct TreeNode\* root, int level) {

if (root == NULL)

return;

if (level == 1)

printf("%d ", root->data);

else if (level > 1) {

printGivenLevel(root->left, level - 1);

printGivenLevel(root->right, level - 1);

}

}

int height(struct TreeNode\* root) {

if (root == NULL)

return 0;

else {

int leftHeight = height(root->left);

int rightHeight = height(root->right);

if (leftHeight > rightHeight)

return (leftHeight + 1);

else

return (rightHeight + 1);

}

}

void printLevelOrder(struct TreeNode\* root) {

int h = height(root);

int i;

for (i = 1; i <= h; i++)

printGivenLevel(root, i);

}

int main() {

struct TreeNode\* root = NULL;

root = insertNode(root, 1);

insertNode(root, 2);

insertNode(root, 3);

insertNode(root, 4);

insertNode(root, 5);

insertNode(root, 6);

insertNode(root, 7);

printf("Level Order traversal of binary tree: ");

printLevelOrder(root);

printf("\n");

return 0;

}

**2. Write a C program to create a binary tree using non-recursive function and display that level wise.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct QueueNode {

struct TreeNode \*treeNode;

struct QueueNode \*next;

};

struct Queue {

struct QueueNode \*front, \*rear;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = queue->rear = NULL;

return queue;

}

void enqueue(struct Queue\* queue, struct TreeNode \*root) {

struct QueueNode\* temp = (struct QueueNode\*)malloc(sizeof(struct QueueNode));

temp->treeNode = root;

temp->next = NULL;

if (queue->rear == NULL) {

queue->front = queue->rear = temp;

return;

}

queue->rear->next = temp;

queue->rear = temp;

}

struct TreeNode\* dequeue(struct Queue\* queue) {

if (queue->front == NULL)

return NULL;

struct TreeNode\* temp = queue->front->treeNode;

struct QueueNode\* tempNode = queue->front;

queue->front = queue->front->next;

if (queue->front == NULL)

queue->rear = NULL;

free(tempNode);

return temp;

}

int isEmpty(struct Queue\* queue) {

return queue->front == NULL;

}

void printLevelOrder(struct TreeNode\* root) {

if (root == NULL)

return;

struct Queue\* queue = createQueue();

enqueue(queue, root);

while (!isEmpty(queue)) {

struct TreeNode\* tempNode = dequeue(queue);

printf("%d ", tempNode->data);

if (tempNode->left != NULL)

enqueue(queue, tempNode->left);

if (tempNode->right != NULL)

enqueue(queue, tempNode->right);

}

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->left = createNode(6);

root->right->right = createNode(7);

printf("Level Order traversal of binary tree: ");

printLevelOrder(root);

printf("\n");

return 0;

}

**3. Write a C program to create a binary tree using array only and display the tree level wise.**

#include <stdio.h>

#include<stdlib.h>

#define MAX\_SIZE 100

struct TreeNode {

int data;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

return newNode;

}

void printLevelOrder(int arr[], int n) {

for (int i = 0; i < n; i++) {

if (arr[i] != -1)

printf("%d ", arr[i]);

}

}

int main() {

int tree[MAX\_SIZE];

int n;

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

printf("Enter the elements of the array (-1 for empty): ");

for (int i = 0; i < n; i++) {

scanf("%d", &tree[i]);

}

printf("Level Order traversal of binary tree: ");

printLevelOrder(tree, n);

printf("\n");

return 0;

}

**4. Write a C program to identify the height of a binary tree.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(struct TreeNode\* root) {

if (root == NULL)

return 0;

else {

int leftHeight = height(root->left);

int rightHeight = height(root->right);

return 1 + max(leftHeight, rightHeight);

}

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->left = createNode(6);

root->right->right = createNode(7);

printf("Height of the binary tree is: %d\n", height(root));

return 0;

}

**5. Write a C program to identify degree of a given node.**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

int degree(struct TreeNode\* root, int key) {

if (root == NULL)

return -1;

if (root->data == key) {

if (root->left != NULL && root->right != NULL)

return 2;

else if (root->left != NULL || root->right != NULL)

return 1;

else

return 0;

}

int leftDegree = degree(root->left, key);

if (leftDegree != -1)

return leftDegree;

return degree(root->right, key);

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->right = createNode(6);

int key;

printf("Enter the node to find its degree: ");

scanf("%d", &key);

int nodeDegree = degree(root, key);

if (nodeDegree == -1)

printf("Node not found in the tree.\n");

else

printf("Degree of node %d is: %d\n", key, nodeDegree);

return 0;

}

**6. Write a C program to count number of leaf node present in a binary tree**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

int countLeafNodes(struct TreeNode\* root) {

if (root == NULL)

return 0;

if (root->left == NULL && root->right == NULL)

return 1;

return countLeafNodes(root->left) + countLeafNodes(root->right);

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->right = createNode(6);

printf("Number of leaf nodes in the binary tree: %d\n", countLeafNodes(root));

return 0;

}

**7. Write a C program to count number of internal node present in a binary tree.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

int countInternalNodes(struct TreeNode\* root) {

if (root == NULL || (root->left == NULL && root->right == NULL))

return 0;

return 1 + countInternalNodes(root->left) + countInternalNodes(root->right);

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->right = createNode(6);

printf("Number of internal nodes in the binary tree: %d\n", countInternalNodes(root));

return 0;

}

**8. Write a C program to count number of node present in a given binary tree using linked list.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

int countNodes(struct TreeNode\* root) {

if (root == NULL)

return 0;

return 1 + countNodes(root->left) + countNodes(root->right);

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->right = createNode(6);

printf("Number of nodes in the binary tree: %d\n", countNodes(root));

return 0;

}

**9. Write a C program to count number of node present in a given binary tree using array.**

#include <stdio.h>

#define MAX\_SIZE 100

struct TreeNode {

int data;

};

int countNodes(int arr[], int n) {

int count = 0;

for (int i = 0; i < n; i++) {

if (arr[i] != -1)

count++;

}

return count;

}

int main() {

int tree[MAX\_SIZE];

int n;

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

printf("Enter the elements of the array (-1 for empty): ");

for (int i = 0; i < n; i++) {

scanf("%d", &tree[i]);

}

printf("Number of nodes in the binary tree: %d\n", countNodes(tree, n));

return 0;

}

**10. Write a C program to count number of siblings present in a binary tree.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

int countSiblings(struct TreeNode\* root, int key) {

if (root == NULL)

return -1;

if ((root->left != NULL && root->left->data == key) ||

(root->right != NULL && root->right->data == key)) {

if (root->left != NULL && root->right != NULL)

return 2;

else if (root->left != NULL || root->right != NULL)

return 1;

else

return 0;

}

int leftSiblings = countSiblings(root->left, key);

if (leftSiblings != -1)

return leftSiblings;

return countSiblings(root->right, key);

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->right = createNode(6);

int key;

printf("Enter the node to find its siblings: ");

scanf("%d", &key);

int siblingsCount = countSiblings(root, key);

if (siblingsCount == -1)

printf("Node not found in the tree or it's the root.\n");

else

printf("Number of siblings of node %d is: %d\n", key, siblingsCount);

return 0;

}

**Assignment-3**

**1. Write a C program to create a binary search tree using recursive function and display that**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (data <= root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

int main() {

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

root = insertNode(root, 30);

root = insertNode(root, 20);

root = insertNode(root, 40);

root = insertNode(root, 70);

root = insertNode(root, 60);

root = insertNode(root, 80);

printf("Binary Search Tree elements in inorder traversal: ");

inorderTraversal(root);

printf("\n");

return 0;

}

**2. Write a C program to create a binary search tree using non-recursive function and display that**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

struct TreeNode\* current = root;

struct TreeNode\* parent = NULL;

while (current != NULL) {

parent = current;

if (data <= current->data) {

current = current->left;

} else {

current = current->right;

}

}

if (data <= parent->data) {

parent->left = createNode(data);

} else {

parent->right = createNode(data);

}

return root;

}

void inorderTraversal(struct TreeNode\* root) {

struct TreeNode\* stack[100];

int top = -1;

if (root == NULL) {

printf("Tree is empty.\n");

return;

}

struct TreeNode\* current = root;

while (current != NULL || top != -1) {

while (current != NULL) {

stack[++top] = current;

current = current->left;

}

current = stack[top--];

printf("%d ", current->data);

current = current->right;

}

}

int main() {

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 70);

insertNode(root, 60);

insertNode(root, 80);

printf("Binary Search Tree elements in inorder traversal: ");

inorderTraversal(root);

printf("\n");

return 0;

}

**3. Write a C program to insert (by using a function) a specific element into an existing binary search tree and then display that.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (data <= root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

int main() {

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 70);

insertNode(root, 60);

insertNode(root, 80);

printf("Binary Search Tree elements in inorder traversal: ");

inorderTraversal(root);

printf("\n");

int elementToInsert = 45;

root = insertNode(root, elementToInsert);

printf("Binary Search Tree elements after inserting %d: ", elementToInsert);

inorderTraversal(root);

printf("\n");

return 0;

}

**4. Write a C program to search an element in a BST and show the result**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data)

{

if (root == NULL) {

root = createNode(data);

return root;

}

if (data <= root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

struct TreeNode\* searchNode(struct TreeNode\* root, int key) {

if (root == NULL || root->data == key)

return root;

if (root->data < key)

return searchNode(root->right, key);

return searchNode(root->left, key);

}

void displaySearchResult(struct TreeNode\* result, int key) {

if (result == NULL) {

printf("Element %d not found in the BST.\n", key);

} else {

printf("Element %d found in the BST.\n", key);

}

}

int main() {

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 70);

insertNode(root, 60);

insertNode(root, 80);

int keyToSearch = 40;

struct TreeNode\* searchResult = searchNode(root, keyToSearch);

displaySearchResult(searchResult, keyToSearch);

return 0;

}

**5. Write a C program to take user name as input and display the sorted sequence of characters using BST.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

struct TreeNode {

char data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(char data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, char data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (data < root->data) {

root->left = insertNode(root->left, data);

} else if (data > root->data) {

root->right = insertNode(root->right, data);

}

return root;

}

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%c", root->data);

inorderTraversal(root->right);

}

}

int main() {

char name[100];

printf("Enter your name: ");

fgets(name, sizeof(name), stdin);

size\_t len = strlen(name);

if (len > 0 && name[len - 1] == '\n') {

name[len - 1] = '\0';

}

for (int i = 0; name[i] != '\0'; i++) {

name[i] = toupper(name[i]);

}

struct TreeNode\* root = NULL;

for (int i = 0; name[i] != '\0'; i++) {

if (isalpha(name[i])) {

root = insertNode(root, name[i]);

}

}

printf("Sorted sequence of characters in your name: ");

inorderTraversal(root);

printf("\n");

return 0;

}

**6. Write a C program to sort a given set of integers using BST.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (data < root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

void inorderTraversalToArray(struct TreeNode\* root, int \*array, int \*index) {

if (root != NULL) {

inorderTraversalToArray(root->left, array, index);

array[(\*index)++] = root->data;

inorderTraversalToArray(root->right, array, index);

}

}

void sortUsingBST(int \*arr, int n) {

struct TreeNode\* root = NULL;

for (int i = 0; i < n; i++) {

root = insertNode(root, arr[i]);

}

int index = 0;

inorderTraversalToArray(root, arr, &index);

}

int main() {

int arr[] = {12, 5, 7, 3, 19, 8, 10};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

sortUsingBST(arr, n);

printf("Sorted array using BST: ");

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

**7. Write a C program to display a BST using In-order, Pre-order, Post-order**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (data <= root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

void preorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

printf("%d ", root->data);

preorderTraversal(root->left);

preorderTraversal(root->right);

}

}

void postorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

postorderTraversal(root->left);

postorderTraversal(root->right);

printf("%d ", root->data);

}

}

int main() {

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 70);

insertNode(root, 60);

insertNode(root, 80);

printf("Inorder traversal: ");

inorderTraversal(root);

printf("\n");

printf("Preorder traversal: ");

preorderTraversal(root);

printf("\n");

printf("Postorder traversal: ");

postorderTraversal(root);

printf("\n");

return 0;

}

**8. Write a C program to Count the number of nodes present in an existing BST and display the highest element present in the BST.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (data <= root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

int countNodes(struct TreeNode\* root) {

if (root == NULL) {

return 0;

}

return 1 + countNodes(root->left) + countNodes(root->right);

}

int findHighestElement(struct TreeNode\* root) {

if (root == NULL) {

printf("BST is empty.\n");

return -1;

}

while (root->right != NULL) {

root = root->right;

}

return root->data;

}

int main() {

struct TreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 70);

insertNode(root, 60);

insertNode(root, 80);

int numNodes = countNodes(root);

printf("Number of nodes in the BST: %d\n", numNodes);

int highestElement = findHighestElement(root);

printf("Highest element in the BST: %d\n", highestElement);

return 0;

}

**9. Write a C program to prove that binary search tree is better than binary tree.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL) {

root = createNode(data);

return root;

}

if (data <= root->data) {

root->left = insertNode(root->left, data);

} else {

root->right = insertNode(root->right, data);

}

return root;

}

int main() {

struct TreeNode\* rootBST = NULL;

struct TreeNode\* rootBT = NULL;

int i;

for (i = 1; i <= 10000; i++) {

rootBST = insertNode(rootBST, i);

}

for (i = 1; i <= 10000; i++) {

rootBT = insertNode(rootBT, rand() % 10000 + 1);

}

return 0;

}

**Assignment-4**

**1. Write a C program to search an element recursively in a binary search tree.**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;

}

struct Node\* search(struct Node\* root, int key) {

if (root == NULL || root->data == key)

return root;

if (root->data < key)

return search(root->right, key);

return search(root->left, key);

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

int key = 60;

struct Node\* result = search(root, key);

if (result != NULL)

printf("%d found in the tree\n", key);

else

printf("%d not found in the tree\n", key);

return 0;

}

**2. Write a C program to delete a node having no child from a binary search tree**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;

}

struct Node\* deleteNode(struct Node\* root, int key) {

if (root == NULL) return root;

if (key < root->data)

root->left = deleteNode(root->left, key);

else if (key > root->data)

root->right = deleteNode(root->right, key);

else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = root->right;

while (temp->left != NULL)

temp = temp->left;

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal before deletion: ");

inorder(root);

printf("\n");

int key = 20;

root = deleteNode(root, key);

printf("Inorder traversal after deletion of %d: ", key);

inorder(root);

printf("\n");

return 0;

}

**3. Write a C program to delete a node having one child from a binary search tree**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;

}

struct Node\* deleteNode(struct Node\* root, int key) {

if (root == NULL) return root;

if (key < root->data)

root->left = deleteNode(root->left, key);

else if (key > root->data)

root->right = deleteNode(root->right, key);

else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = root->right;

while (temp->left != NULL)

temp = temp->left;

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal before deletion: ");

inorder(root);

printf("\n");

int key = 30;

root = deleteNode(root, key);

printf("Inorder traversal after deletion of %d: ", key);

inorder(root);

printf("\n");

return 0;

}

**4. Write a C program to delete a node having two child from a binary search tree**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

struct Node\* minValueNode(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

struct Node\* deleteNode(struct Node\* root, int key) {

if (root == NULL) return root;

if (key < root->data)

root->left = deleteNode(root->left, key);

else if (key > root->data)

root->right = deleteNode(root->right, key);

else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = minValueNode(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal before deletion: ");

inorder(root);

printf("\n");

int key = 50;

root = deleteNode(root, key);

printf("Inorder traversal after deletion of %d: ", key);

inorder(root);

printf("\n");

return 0;

}

**5. Write a C program to delete a node from a binary search tree**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

struct Node\* minValueNode(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

struct Node\* deleteNode(struct Node\* root, int key) {

if (root == NULL) return root;

if (key < root->data)

root->left = deleteNode(root->left, key);

else if (key > root->data)

root->right = deleteNode(root->right, key);

else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = minValueNode(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal before deletion: ");

inorder(root);

printf("\n");

int key = 50;

root = deleteNode(root, key);

printf("Inorder traversal after deletion of %d: ", key);

inorder(root);

printf("\n");

return 0;

}

**Assignment-5**

**1. Write a C program to store the following Graph using Adjacency Matrix & display that.**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_VERTICES 10

int graph[MAX\_VERTICES][MAX\_VERTICES] = {0};

int main() {

int num\_vertices = 6;

graph['P' - 'A']['Q' - 'A'] = 1;

graph['P' - 'A']['T' - 'A'] = 7;

graph['P' - 'A']['S' - 'A'] = 6;

graph['Q' - 'A']['S' - 'A'] = 4;

graph['Q' - 'A']['R' - 'A'] = 1;

graph['T' - 'A']['U' - 'A'] = 2;

graph['S' - 'A']['T' - 'A'] = 3;

graph['S' - 'A']['U' - 'A'] = 2;

graph['R' - 'A']['S' - 'A'] = 2;

graph['R' - 'A']['U' - 'A'] = 1;

printf("Adjacency Matrix Representation of the Graph:\n");

printf(" ");

for (int i = 0; i < num\_vertices; ++i)

printf("%c ", 'A' + i);

printf("\n");

for (int i = 0; i < num\_vertices; ++i) {

printf("%c ", 'A' + i);

for (int j = 0; j < num\_vertices; ++j) {

printf("%d ", graph[i][j]);

}

printf("\n");

}

return 0;

}

**2. Write a C program to store the following Graph using Adjacency List & display that.**

#include <stdio.h>

#include <stdlib.h>

struct Node {

char vertex;

int weight;

struct Node\* next;

};

struct Node\* createNode(char vertex, int weight) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->vertex = vertex;

newNode->weight = weight;

newNode->next = NULL;

return newNode;

}

void addEdge(struct Node\* adjList[], char src, char dest, int weight) {

struct Node\* newNode = createNode(dest, weight);

newNode->next = adjList[src - 'P'];

adjList[src - 'P'] = newNode;

}

void displayGraph(struct Node\* adjList[], int numVertices) {

printf("Adjacency List Representation of the Graph:\n");

for (int i = 0; i < numVertices; ++i) {

printf("%c -> ", 'P' + i);

struct Node\* temp = adjList[i];

while (temp) {

printf("%c (%d) ", temp->vertex, temp->weight);

temp = temp->next;

if (temp)

printf("-> ");

}

printf("\n");

}

}

int main() {

int numVertices = 6;

struct Node\* adjList[numVertices];

for (int i = 0; i < numVertices; ++i)

adjList[i] = NULL;

addEdge(adjList, 'P', 'Q', 1);

addEdge(adjList, 'P', 'T', 7);

addEdge(adjList, 'P', 'S', 6);

addEdge(adjList, 'Q', 'S', 4);

addEdge(adjList, 'Q', 'R', 1);

addEdge(adjList, 'T', 'U', 2);

addEdge(adjList, 'S', 'T', 3);

addEdge(adjList, 'S', 'U', 2);

addEdge(adjList, 'R', 'S', 2);

addEdge(adjList, 'R', 'U', 1);

displayGraph(adjList, numVertices);

return 0;

}

**3. Write a C program to count number of vertices and edges present in a graph.**

#include <stdio.h>

#define MAX 10

int main() {

int graph[MAX][MAX], vertices, edges, i, j;

printf("Enter the number of vertices in the graph: ");

scanf("%d", &vertices);

printf("Enter the adjacency matrix representation of the graph (0 for no edge, 1 for edge):\n");

for (i = 0; i < vertices; i++) {

for (j = 0; j < vertices; j++) {

scanf("%d", &graph[i][j]);

}

}

edges = 0;

for (i = 0; i < vertices; i++) {

for (j = 0; j < vertices; j++) {

if (graph[i][j] == 1) {

edges++;

}

}

}

printf("Number of vertices: %d\n", vertices);

printf("Number of edges: %d\n", edges / 2);

return 0;

}

**4. Write a C program to detect a cycle in a graph.**

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

int adj[MAX][MAX];

int visited[MAX];

int isCyclicUtil(int v, int parent, int numVertices) {

visited[v] = 1;

for (int i = 0; i < numVertices; i++) {

if (adj[v][i]) {

if (!visited[i]) {

if (isCyclicUtil(i, v, numVertices))

return 1;

} else if (i != parent)

return 1;

}

}

return 0;

}

int isCyclic(int numVertices) {

for (int i = 0; i < numVertices; i++)

visited[i] = 0;

for (int i = 0; i < numVertices; i++) {

if (!visited[i]) {

if (isCyclicUtil(i, -1, numVertices))

return 1;

}

}

return 0;

}

int main() {

int numVertices, numEdges;

printf("Enter the number of vertices: ");

scanf("%d", &numVertices);

printf("Enter the number of edges: ");

scanf("%d", &numEdges);

for (int i = 0; i < numVertices; i++) {

for (int j = 0; j < numVertices; j++) {

adj[i][j] = 0;

}

}

printf("Enter the edges:\n");

for (int i = 0; i < numEdges; i++) {

int u, v;

scanf("%d %d", &u, &v);

adj[u][v] = 1;

adj[v][u] = 1;

}

if (isCyclic(numVertices))

printf("Graph contains cycle.\n");

else

printf("Graph doesn't contain cycle.\n");

return 0;

}

**5. Write a C program to identify number of odd degree vertices and number of even degree vertices in a graph.**

#include <stdio.h>

#define MAX 100

int adj[MAX][MAX];

int degree[MAX];

void initialize(int numVertices) {

for (int i = 0; i < numVertices; i++) {

degree[i] = 0;

for (int j = 0; j < numVertices; j++) {

adj[i][j] = 0;

}

}

}

void addEdge(int u, int v) {

adj[u][v] = 1;

adj[v][u] = 1;

degree[u]++;

degree[v]++;

}

int countOddDegreeVertices(int numVertices) {

int count = 0;

for (int i = 0; i < numVertices; i++) {

if (degree[i] % 2 != 0) {

count++;

}

}

return count;

}

int countEvenDegreeVertices(int numVertices) {

int count = 0;

for (int i = 0; i < numVertices; i++) {

if (degree[i] % 2 == 0) {

count++;

}

}

return count;

}

int main() {

int numVertices, numEdges;

printf("Enter the number of vertices: ");

scanf("%d", &numVertices);

printf("Enter the number of edges: ");

scanf("%d", &numEdges);

initialize(numVertices);

printf("Enter the edges:\n");

for (int i = 0; i < numEdges; i++) {

int u, v;

scanf("%d %d", &u, &v);

addEdge(u, v);

}

int oddDegreeVertices = countOddDegreeVertices(numVertices);

int evenDegreeVertices = countEvenDegreeVertices(numVertices);

printf("Number of odd degree vertices: %d\n", oddDegreeVertices);

printf("Number of even degree vertices: %d\n", evenDegreeVertices);

return 0;

}

**6. Write a C program to check whether a given graph is complete or not.**

#include <stdio.h>

#define MAX 100

int adj[MAX][MAX];

int isCompleteGraph(int numVertices) {

for (int i = 0; i < numVertices; i++) {

for (int j = 0; j < numVertices; j++) {

if (i != j && adj[i][j] == 0)

return 0;

}

}

return 1;

}

int main() {

int numVertices, numEdges;

printf("Enter the number of vertices: ");

scanf("%d", &numVertices);

printf("Enter the number of edges: ");

scanf("%d", &numEdges);

for (int i = 0; i < numVertices; i++) {

for (int j = 0; j < numVertices; j++) {

adj[i][j] = 0;

}

}

printf("Enter the edges:\n");

for (int i = 0; i < numEdges; i++) {

int u, v;

scanf("%d %d", &u, &v);

adj[u][v] = 1;

adj[v][u] = 1;

}

if (isCompleteGraph(numVertices))

printf("The graph is complete.\n");

else

printf("The graph is not complete.\n");

return 0;

}

**Assignment-6**

**1. Write a C program to traverse the following graph using Depth First Search (DFS) algorithm**

#include <stdio.h>

#include <stdlib.h>

#define MAX 6

struct Edge {

int dest;

int weight;

struct Edge\* next;

};

struct Vertex {

int visited;

struct Edge\* edges;

};

struct Graph {

struct Vertex vertices[MAX];

};

struct Stack {

int items[MAX];

int top;

};

void initGraph(struct Graph\* graph) {

for (int i = 0; i < MAX; ++i) {

graph->vertices[i].visited = 0;

graph->vertices[i].edges = NULL;

}

}

void addEdge(struct Graph\* graph, int src, int dest, int weight) {

struct Edge\* newEdge = (struct Edge\*)malloc(sizeof(struct Edge));

newEdge->dest = dest;

newEdge->weight = weight;

newEdge->next = graph->vertices[src].edges;

graph->vertices[src].edges = newEdge;

}

void push(struct Stack\* stack, int vertex) {

stack->items[++stack->top] = vertex;

}

int pop(struct Stack\* stack) {

return stack->items[stack->top--];

}

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

void dfs(struct Graph\* graph, int start) {

struct Stack stack;

stack.top = -1;

push(&stack, start);

while (!isEmpty(&stack)) {

int current = pop(&stack);

if (!graph->vertices[current].visited) {

printf("%c ", current + 'P');

graph->vertices[current].visited = 1;

struct Edge\* edge = graph->vertices[current].edges;

while (edge != NULL) {

if (!graph->vertices[edge->dest].visited) {

push(&stack, edge->dest);

}

edge = edge->next;

}

}

}

}

int main() {

struct Graph graph;

initGraph(&graph);

addEdge(&graph, 0, 1, 1); // P -> Q = 1

addEdge(&graph, 0, 2, 7); // P -> T = 7

addEdge(&graph, 0, 3, 6); // P -> S = 6

addEdge(&graph, 1, 3, 4); // Q -> S = 4

addEdge(&graph, 1, 4, 1); // Q -> R = 1

addEdge(&graph, 2, 5, 2); // T -> U = 2

addEdge(&graph, 3, 2, 3); // S -> T = 3

addEdge(&graph, 3, 5, 2); // S -> U = 2

addEdge(&graph, 4, 3, 2); // R -> S = 2

addEdge(&graph, 4, 5, 1); // R -> U = 1

printf("Depth First Traversal starting from P: ");

dfs(&graph, 0); // Start DFS from P (index 0)

printf("\n");

return 0;

}

**2. Write a C program to traverse the following graph using Breadth First Search (BFS) algorithm**

#include <stdio.h>

#include <stdlib.h>

#define MAX 6

struct Edge {

int dest;

int weight;

struct Edge\* next;

};

struct Vertex {

int visited;

struct Edge\* edges;

};

struct Graph {

struct Vertex vertices[MAX];

};

struct Queue {

int items[MAX];

int front;

int rear;

};

void initGraph(struct Graph\* graph) {

for (int i = 0; i < MAX; ++i) {

graph->vertices[i].visited = 0;

graph->vertices[i].edges = NULL;

}

}

void addEdge(struct Graph\* graph, int src, int dest, int weight) {

struct Edge\* newEdge = (struct Edge\*)malloc(sizeof(struct Edge));

newEdge->dest = dest;

newEdge->weight = weight;

newEdge->next = graph->vertices[src].edges;

graph->vertices[src].edges = newEdge;

}

void enqueue(struct Queue\* queue, int vertex) {

queue->rear++;

queue->items[queue->rear] = vertex;

}

int dequeue(struct Queue\* queue) {

int item = queue->items[queue->front];

queue->front++;

return item;

}

int isEmpty(struct Queue\* queue) {

return queue->rear == queue->front - 1;

}

void bfs(struct Graph\* graph, int start) {

struct Queue queue;

queue.front = 0;

queue.rear = -1;

graph->vertices[start].visited = 1;

enqueue(&queue, start);

while (!isEmpty(&queue)) {

int current = dequeue(&queue);

printf("%c ", current + 'P');

struct Edge\* edge = graph->vertices[current].edges;

while (edge != NULL) {

int dest = edge->dest;

if (!graph->vertices[dest].visited) {

graph->vertices[dest].visited = 1;

enqueue(&queue, dest);

}

edge = edge->next;

}

}

}

int main() {

struct Graph graph;

initGraph(&graph);

addEdge(&graph, 0, 1, 1); // P -> Q = 1

addEdge(&graph, 0, 2, 7); // P -> T = 7

addEdge(&graph, 0, 3, 6); // P -> S = 6

addEdge(&graph, 1, 3, 4); // Q -> S = 4

addEdge(&graph, 1, 4, 1); // Q -> R = 1

addEdge(&graph, 2, 5, 2); // T -> U = 2

addEdge(&graph, 3, 2, 3); // S -> T = 3

addEdge(&graph, 3, 5, 2); // S -> U = 2

addEdge(&graph, 4, 3, 2); // R -> S = 2

addEdge(&graph, 4, 5, 1); // R -> U = 1

printf("Breadth First Traversal starting from P: ");

bfs(&graph, 0);

printf("\n");

return 0;

}